Experiments in the Concurrent Computation of Spatial Association on the KSR-1

Richard J. Marciano[1]
Marc P. Armstrong[2]
Claire E. Pavlik[3]

January 10, 1993

richard-marciano@uiowa.edu
Fencing Technologies
phone & fax: (612) 926-7955

marc-armstrong@uiowa.edu
claire-pavlik@uiowa.edu
Department of Geography
phone: (319) 335-1050
fax: (319) 335-2725

1. Gerard P. Weeg Computer Center, University of Iowa, and Fencing Technologies, Minneapolis, Minnesota.

2. Departments of Geography and Computer Science, and Program in Applied Mathematical and Computational Sciences, University of Iowa.

3. Department of Geography, University of Iowa.

Acknowledgments: This work was supported in part by the National Science Foundation (SES-9024278). The Cornell National Supercomputing Facility provided access to the KSR-1 computer system that was used to prepare the results reported in this paper. We wish to thank Arthur Getis for kindly providing the original serial version of the code used in the analyses. Many thanks to Dan Dwyer, former Smart Node Program Coordinator at Cornell, and to Mike Hammill, current Smart Node Program Coordinator, for their technical insights and parallel programming assistance.

TABLE OF CONTENTS

Experiments in the Concurrent Computation of Spatial Association on the KSR-1

Abstract

Spatial association measures, when computed for large datasets, have significant computational requirements. Parallel processing is one way to address these requirements. The design of parallel programs, however, requires careful planning since many factors under programmer control affect the efficiency of the resulting computations. In this paper, two strategies of parallel processing are described using as an illustration a measure of spatial association, **G**(d). An evaluation is made of the efficiency of the parallel implementations by varying the problem size and the number of processors used in parallel. The results obtained indicate that parallel processing can currently enable analysts to work in near-real-time with problems that range in the tens of thousands of observations; such problems require less than two minutes of execution time on the KSR-1. Also superlinear speedups of almost 1500 are obtained for the *computation phase* of the problem.

Key Words: Parallel processing, spatial statistics, KSR, MPP

1.0 INTRODUCTION

The need to incorporate spatial dependencies in geographical models has a long history of acknowledgment in theory, but neglect in practice. Gould (1973), for example, questioned the application of classical inferential statistical methods to geographical data, while Cliff and Ord (1973) demonstrated that spatial dependencies, with their underlying assumption of independent observations, can throw the interpretation of regression results into question. Though Anselin and Griffith (1988) and Getis and Ord (1992), among others, have recently refocussed attention on the measurement of spatial dependency, for the past two decades practitioners have often ignored its measurement for several reasons:

(1) Researchers have often lacked the training needed to perform spatial dependency analyses. As a result of the recent publication of tutorials (e.g. Goodchild, 1986; Griffith, 1987; Odland, 1988) and more comprehensive texts on spatial statistics (e.g. Cressie, 1992), this problem is becoming less significant.

(2) Easy to use software that evaluates datasets for spatial dependencies has not been widely available. This problem is also becoming less significant as researchers have demonstrated that commercial statistical software can be applied to geographical problems (Griffith, 1987) and have developed programs that are specifically designed to evaluate data for spatial dependencies (Anselin, 1991). Moreover, many GIS software packages now provide a means for measuring spatial dependencies.

(3) The application of many measures of spatial dependence to large realistic problems has not been feasible because of the numerical intensity of the required computations (e.g. Griffith, 1990).

The purpose of this paper is to address this third issue. We illustrate the magnitude of these computational requirements and present the results of a series of computational experiments that were developed to reduce execution times when a recently derived (Getis and Ord, 1992) measure of spatial dependency, **G**(d), is applied to large datasets. The experiments are designed to show how a straightforward translation of statistical equations into computer code can be improved by decomposing the problem into a set of processes that are executed concurrently using a parallel supercomputer.

1.1 Background: G Statistic

G(d) measures the spatial association of positive-valued point data in the plane; it does not provide a single, global assessment of spatial structure but, instead, is reported as a series of measures calculated for user-specified distances. Getis and Ord (1992) state that this intra-zonal focus is important when data have unequal local averages and when values are related by the distance between observations rather than by absolute location. These characteristics are often found in spatial data collections (Cliff and Ord, 1981; Oliver *et al.*, 1989).

G(d) can be summarized in the following way: Define a dataset of *n* observations each described by coordinates and a positive, real-valued variable, **Z**. Since the **G**(d) statistic summarizes the characteristics of the **Z**-values contained in circles of varying sizes, select an observation, located at site *i*, and center a circle of radius *d* upon it. Compute the sum of the products of site *i*'s **Z**-value with the **Z**-value of the other sites within the circle, and accumulate these products. Repeat the procedure for each site using circles of the same radius, then sum the accumulated values for all sites. Finally, normalize this result by dividing it by the double-sum of all different site $z_i \cdot z_j$-products. Repeat the process for all specified radius values.

1.2 Computational Requirements

The calculation of **G**(d) requires two main steps, referred to below as *initialization* and *computation*, that each requires substantial amounts of computation. In the *initialization phase*, vectors X, Y, Z are read, interpoint distances are computed and the results are stored in a matrix (called "DIST").

```
      DO 2 J=1,NOBS
            DO 2 I=1,NOBS
                  IF (I .NE. J) THEN
                        XDIST = X(I) - X(J)
                        YDIST = Y(I) - Y(J)
                        DIST(I,J) = SQRT ((XDIST * XDIST) + (YDIST * YDIST))
                  END IF
2     CONTINUE
```

The number of calculated distances increases with the square of the number of observations in this example, though simple arithmetic assignments can reduce the amount of computation in this part of the program provided that DIST(I,J) = DIST(J,I). Thus for a 10,000 observation analysis, approximately 10^8 distance calculations (N^2-N) are required. If the symmetry of the matrix is exploited to reduce computation 49,995,000 distances (N^2-N)/2) must be computed.

In the *computation stage*, for each d-value, DIST is used to ensure that only points that fall within a window specified by the current value of the radius are included in the computation of **G**(d). This process is repeated until the radius reaches the desired maximum value of *d*. Thus, the total amount of computation required depends partly on the number of radii for which **G**(d) is computed. In general, the *computation time* of

this phase increases as a function of the size of the problem and the number of radii for which **G**(d) is computed.

The following fragment of code illustrates the computation stage where d is called STEP, and where a G value corresponding to that particular STEP value is computed (G = ZS / ZSUM):

```
      STEP = 0.
70    STEP = STEP + (MAX / INC)
            ZSUM = 0.
            ZS = 0.
            WT = 0.
            DO 3 J=1,NOBS
                  WA = 0
                  DO 3 I=1,NOBS
                        IF (I .NE. J) THEN
                              ZSUM = ZSUM + (Z(I) * Z(J))
                              IF (DIST(I,J) .LE. STEP) THEN
                                    ZS = ZS + (Z(I) * Z(J))
                                    WT = 1 + WT
                                    WA = WA + 1
                                    W(J) = WA
                              END IF
                        END IF
3           CONTINUE
            G = ZS / ZSUM
            S2 = 0.
            DO 4 J=1,NOBS
                  S2 = S2 + W(J)*4*W(J)
4           CONTINUE

C           ..........  MORE CODE LEFT OUT ..........

      IF (STEP .EQ. MAX) GO TO 80
      GO TO 70
80    CONTINUE
```

To demonstrate its computational requirements, a serial implementation of code that calculates **G**(d) was ported to three platforms: a **PC** (20 Mhz IBM PS2/55 SX 386 processor with no floating-point accelerator), a **mainframe** (Encore Multimax), and a **supercomputer** (KSR-1). Two important limitations were examined: the maximum problem size and runtime.

With 256 observations, the executable code generated on the PC used close to 300KB; when the number of observations was increased to 384, the PC's memory capacity (640KB) was exceeded. Consequently, the largest problem size that could be run on a PC was approximately 300 points. When the problem size was reduced to 256

observations and **G**(d) was calculated for 20 radii increasing in increments of 5 units from 5 to 100, the PC required nearly 20 minutes to calculate the results (Table 1). While faster PCs are available and memory restrictions are being overcome with the provision of new operating systems, PC-class computers preclude interactive analyses for the **G**(d) statistic and limit the size of the analyses that may be successfully undertaken.

Table 1. Execution times (seconds) for a 256 point data set.

	386 PC	Encore	KSR-1
Initialization	87.0	3.60	0.04
Computation	1106.0	29.30	1.62
Total Time	**1193.0**	**32.90**	**1.66**

As one might expect, the mainframe class Encore computer is capable of handling larger datasets. The largest sized problem that could be accommodated, however, is still fairly small with 1600 points. Also note from Table 1 that the Encore required more than half a minute of execution time for the small 256 point dataset, while the total runtime for the KSR-1 was less than two seconds. These preliminary analyses demonstrate that high performance computing is required for two reasons: first, to handle large problems (we would like to analyze data sets that contain more than 10,000 observations) and, second, to obtain runtimes that enable analysts to perform interactive exploratory analyses of large spatial data sets. Consequently, all further analyses are reported using the KSR-1 parallel supercomputer. In section 2, we briefly describe the characteristics of this high performance computer. Following that, we describe two different implementations of the **G**(d) algorithm and evaluate their performance in a series of processing experiments that exploit the architectural characteristics of the KSR-1.

2.0 THE KSR-1 ENVIRONMENT

The KSR-1 is a massively parallel computer composed of 64-bit RISC-style superscalar processors with independent integer and floating point units; the floating point units also have independent, pipelined adder/multipliers. Each processor used by the KSR-1 is called a cell and runs at 20-MHz for a peak performance rate of 40 Megaflops per cell; the number of cells may range from 8 to 1088. The model we used

at the Cornell National Supercomputer Facility was a 64-cell machine, structured as a ring of 2 rings of 32 cells each (see section 2.1 for details).

2.1 Hardware Description

The KSR-1 is a hybrid architecture in the sense that both the shared-memory and distributed-memory computing models are used. At first glance, it may seem to be a shared-memory machine in which programs and data are mapped into a single large virtual address space. However, physical storage is composed of a collection of local caches, one for each processor (or cell), each capable of storing 32 MBytes of information. Each cell also has an additional, faster 0.5 MB hardware cache consisting of a 256KB data cache, called the subcache, and a 256KB instruction cache. Data is stored in units of pages and subpages; a local cache can store 2K pages, each divided into 128 subpages of 128 bytes (2K x 128 x 128 = 32MB).

Local caches are interconnected through a mechanism called the search engine. If a cell must access memory located outside its own cache, it generates a request to a search engine that interconnects the local caches and provides routing and directory services for all caches. It is this characteristic of the architecture that lets the distributed local caches behave logically as a single, shared address space. The combination of local caches and the search engine is implemented as a hierarchy of search groups. If a cell requests access to a memory location that is not in its subcache, then the search engine attempts to fetch data from the local cache. If the required data are not in the local cache, then the search engine generates a request that is placed into an open slot on the local communications ring (Ring:0). This request (a packet) is passed to the next cell on the ring, which then tries to match the packet against its local cache by consulting the cell's local-cache directory. If the match is successful, the packet is extracted from the ring, reinserted with the data, and continues around the ring back to the cell that generated the request. One of the Ring:0 cells is a special cell that connects to the next higher level (Ring:1) in the hierarchy. This cell includes a directory comprising all the entries in Ring:0's local-cache directories. When a packet reaches this cell, if the requested address is not present in the Ring:0 directory, then the packet is routed to the next higher level, Ring:1, which connects up to 34 other Ring:0s.

The aggregate capacity of each level of the cache hierarchy and the latency (in processor cycles) between the processor and each cache level is shown in Table 2.

Latency times are very important in this architecture because when a cell generates a memory request, processing cannot proceed until the needed data arrives. Given that latencies increase when non-local memory references are encountered, to be efficient programs must minimize access to memory locations that are far, in an architectural sense, from the local cache. Because performance is linked to an effective exploitation of the cache hierarchy, the characteristics of a problem must be matched to the characteristics of the hardware for maximum performance.

Table 2. Cache hierarchy characterization.

Location of cache	*Local cache capacity*	*Latency-clock cycles*
Local subcache	256KB	2
Local cache	32MB	18
Search Group:0	1GB	175
Search Group:1	34GB	600
Page fault	N.A.	400,000 (~20 msecs)

Source: Kendall Square Research, 1992b.

2.2 Parallel Programming Software Support

The underlying mechanism used to execute parallel constructs in this programming environment is a pthread, a single sequential flow of control within a process (Kendall Square Research, 1992a). A FORTRAN program creates, synchronizes, schedules, and cancels pthreads using library calls or compiler directives that invoke calls to the runtime library. There are four main high level FORTRAN parallel constructs that are normally used to implement programs on the KSR-1:

(1) Parallel regions -- multiple instantiations of a code segment are executed in parallel.

(2) Parallel sections -- multiple code segments are executed in parallel.

(3) Tile families -- loops are executed in parallel (often used with affinity regions). Loops can be tiled automatically, semi-automatically, or manually by the programmer.

(4) Affinity regions -- all tile families in an affinity region are executed by the same team of pthreads, using the same pthread-to-tile assignments. This ensures that data movement is minimized and cache performance is improved.

These constructs are implemented using compiler directives that start with **C*KSR*** to improve portability and readability of code. *Tile,* for example, is a commonly used directive that divides loop iterations into blocks that run on different threads. The tiling strategy used by a parallel program affects the partitioning and assignment of the workload to the available processors. The tile directive has the following syntax:

c*ksr* tile (**index_list** [options])
..... loop to be tiled .....
c*ksr* end tile

The options used in our parallel spatial statistics implementation are:

- **index_list**
  specifies the index variables of the loops to be tiled.
- **private** = (variable_list)
  specifies variables for which each thread has its own "private" copy.
- **reduction** = (variable_list)
  specifies a list of variables which are reduction variables for the tile family. Reduction variables accumulate values for separate tiles that need to be pooled into a single value at the end of the parallel construct.
- **strategy** = (slice | mod | grab | wave)
  specifies the strategy to be used to order the execution of tiles in a tile family. We used the **slice** strategy in which loop iterations are divided into a number of tiles equal to the number of threads executing the tile family. Each pthread executes one tile, and each tile is the same size. This facilitates load balancing among the processors.
- **teamid** = (team_id)
  specifies the ID of the team used to execute the tile family.

3.0 IMPLEMENTATIONS USING A SINGLE PROCESSOR

Two implementations of **G**(d) are investigated in this section. The first, a "naive" implementation, is a straightforward translation of the equations reported by Getis and Ord (1992) into a series of FORTRAN statements that are executed using one processor. This serves as a serial benchmark against which the parallel results reported in the

following section can be compared. The second, what we call a "sorting" implementation, is intended to improve performance by restructuring the way that statistical indices are calculated.

3.1 Serial Algorithm Modifications

In the "sorting" modification, the structure of the problem is exploited: distance values in each column of the matrix, DIST, are sorted in ascending order using a *shellsort* algorithm given in Sedgewick (1990), and a matrix of *zi*•*zj* products, ZZ, is sorted into the same order by columns. The distance matrix, DIST, is then used to construct a "mask" of the *zi*•*zj* products that fall within the initial d-radius and successive, incremental rings about each data point.

X	40.8	77.9	25.9	42.1	53.1	4.0	58.6	53.2	39.7
Y	66.7	23.5	48.8	34.6	91.5	49.2	30.8	27.5	57.9
Z	74.7	30.6	62.6	78.6	25.0	18.4	70.1	66.1	87.5

DIST

0.0	57.0	23.4	32.2	27.7	40.8	40.2	41.2	8.9
57.0	0.0	57.9	37.5	72.4	78.2	20.6	25.0	51.4
23.4	57.9	0.0	21.6	50.7	21.8	37.4	34.7	16.6
32.2	37.5	21.6	0.0	58.0	40.8	16.9	13.2	23.5
27.7	72.4	50.7	58.0	0.0	64.9	61.0	64.0	36.2
40.8	78.2	21.8	40.8	64.9	0.0	57.6	53.8	36.8
40.2	20.6	37.4	16.9	61.0	57.6	0.0	6.3	33.0
41.2	25.0	34.7	13.2	64.0	53.8	6.3	0.0	33.3
8.9	51.4	16.6	23.5	36.2	36.8	33.0	33.3	0.0

STEP = 30

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8.9	20.6	16.6	13.2	27.7	21.8	6.3	6.3	8.9
23.4	25.	21.6	16.9	36.2	36.8	16.9	13.2	16.6
27.7	37.5	21.8	21.6	50.7	40.8	20.6	25.0	23.5
32.2	51.4	23.4	23.5	58.0	40.8	33.0	33.3	33.0
40.2	57.0	34.7	32.2	61.0	53.8	37.4	34.7	33.3
40.8	57.9	37.4	37.5	64.0	57.6	40.2	41.2	36.2
41.2	72.4	50.7	40.8	64.9	64.9	57.6	53.8	36.8
57.0	78.2	57.9	58.0	72.4	78.2	61.0	64.0	51.4

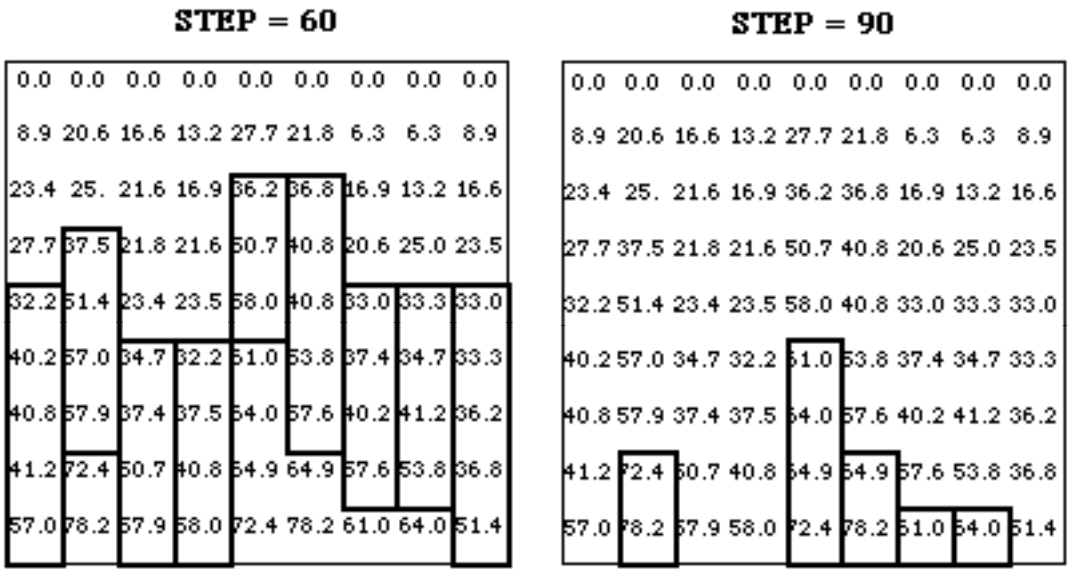

Figure 1. "Sorting" algorithm: using logical masks to reduce calculations in the *computation phase*.

Figure 1 shows the X and Y coordinates for 9 points, their corresponding Z values, the resulting DIST interpoint distance matrix, and how a column-sorted DIST matrix is used to create successive "masks" that are applied to the ZZ matrix. In this example, we assume a maximum radius of 90 units and show the execution for 3 radius increments of 30 units, resulting in STEP values of 30, 60, and 90. For a particular radius value, a highlighted rectangle defines a mask that is applied to the corresponding locations in ZZ. Only those elements of ZZ that fall within the mask are summed for each particular STEP.

This mask is implemented using an array, called **startindex**, that for each radius value, contains the beginning positions of each column. The startindex array for this example is shown in Figure 2.

Initially:

2	2	2	2	2	2	2	2	2

After radius circle d=30:

5	4	6	6	3	3	5	5	5

After radius circle d=60:

10	8	10	10	6	8	9	9	10